Testing avec Angular

Dans ce document, nous allons voir comment tester efficacement une application Angular. Angular est un Framework JavaScript très complet qui inclus deux systèmes de testing : Karma pour les tests unitaires et Protractor pour les tests e2e.

- Les **tests unitaires** : sont utilisés pour tester si une fonctionnalité répond correctement au besoin sans spécifier un contexte particulier. En effet elles sont censées fonctionner de la même façon peu importe le contexte.
- Les **tests end-to-end** : sont utilisés pour vérifier l'interopérabilité des différentes fonctionnalités. Nous allons par exemple tester les différentes navigations ou encore qu'on ne peut pas soumettre un formulaire si les champs requis ne sont pas tous replis…

Best practice

Créer un hook Git « pre-commit.sh » afin de ne pas pouvoir commiter tant que les tests ne valident pas votre code :

```
#!/bin/sh

npm test &&
npm run lint
```

Pour plus de facilité d'installation du hook, ajouter ce script dans votre « package.json » :

```
    "setup-hooks": "ln -s git-hooks/pre-commit.sh .git/hooks/pre-commit"
```

Vous pouvez alors l'installer en utilisant la commande « npm run setup-hooks ».

Table des matières

Test unitaire

Il s'agit de tous les fichiers « .spec.ts » se situant à côté des éléments à tester.
Ces tests sont exécutables via la commande « ng test » (préférez « npm run test »).

Service business

Si le service ne nécessite pas ou peu de dépendances, le plus simple et efficace est d'utiliser uniquement les outils de Karma. Pour se faire, il faut d'abord créer une instance du service :

```
import { MetaService } from './meta.service';

describe('Service: Meta', () => {
  let service: MetaService;

  beforeEach(() => {
    service = new MetaService();
  });

  ...

});
```

Il est tout aussi simple de tester le retour d'une fonction :

```
  it('#getHost should return http://test.net', () => {
    expect(service.getHost('http://test.net/bonjour')).toBe('http://test.net');
  });
```

Ou encore un appel asynchrone à l'aide du paramètre « done » qui permet de notifier à Karma quand le test peut-être considéré comme terminé :

```
  it('#getObservableValue should http://test.net from observable',
    (done: DoneFn) => {
    service.getHostAsync('http://test.net/bonjour').subscribe(value => {
      expect(value).toBe('http://test.net');
      done();
    });
  });
```

Si vous avez besoin d'injecter une dépendance, vous pouvez simplement créer une instance de celle-ci et la fournir à votre service :

```
beforeEach(() => {
    service = new MetaService(new MyDependency());
  });
```

Ou utiliser un « Fake » :

```
class MyFakeDependency {
  public sameMethodAsInDependency() {
    ...
  }
}

describe('Service: Meta', () => {
  let service: MetaService;

  beforeEach(() => {
    service = new MetaService(new MyFakeDependency() as MyDependency);
```

```
  });
```

Si le service dépend d'autres services qui ont eux aussi d'autres dépendances, leurs instanciations risque de vite devenir pénible. Nous allons donc utiliser l'outil le plus intéressant mis à disposition par Angular pour les tests unitaires : le TestBed. Celui-ci sert à simuler un NgModule avec la config que vous lui passez (providers, imports, declarations…) :

```
  let service: MetaService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MetaService,
        MyDependency,
        ...
      ]
    });
    service = TestBed.get(MetaService);
  });
```

Service web

Pour tester les mappings que vous auriez fait au retour d'un appel service, il est préférable de les mocker afin de ne pas avoir de problèmes liés à la connectivité. Cela est possible à l'aide du module HttpClientTestingModule qui comprend une version de l'HttpClient spécialement conçue pour les tests :

```
describe('Service: Users', () => {

  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UsersService]
    });
    httpTestingController = TestBed.get(HttpTestingController);
  });

  it('should get all users', inject([UsersService], (service: UsersService) => {
    const serviceResponse = { users: [{ name: 'hello' }] };
    const mappingResult = [{ name: 'hello' }];

    // Lancer l'appel et vérifier que la réponse est bien celle attendue
    service.getAll().subscribe(response =>
expect(response).toEqual(mappingResult));

    // Ecouter les appels sur l'URL "/users" (celle exécutée à la ligne précédente)
    const req = httpTestingController.expectOne('/users');
    // Vérifier que la méthode est bien celle attendue
    expect(req.request.method).toEqual('GET');
    // Envoyer des données mockées au service
    req.flush(serviceResponse);
```

```
  }));

  afterEach(() => {
    // Vérifier que tous les appels services lancés pendant chaque test ont bien
été vérifié
    httpTestingController.verify();
  });

});
```

Il est possible de tester plusieurs appels avec des réponses différentes en remplaçant « expectOne » par la méthode « match » qui va alors renvoyer une liste de requêtes à la place d'une seule :

```
    const requests = httpTestingController.match('/users');
    expect(requests.length).toEqual(2);
    requests[0].flush({ users: [] });
    requests[1].flush({ users: [{ name: 'hello' }] });
```

Enfin pour tester la réaction aux erreurs, il est également possible d'en mocker en spécifiant un code d'erreur ou non :

```
    // Simulation d'une erreur ayant un code d'erreur
    req.flush('message', {status: 404, statusText: 'Not Found'});
    // Simulation d'une erreur n'ayant pas de code d'erreur
    req.error(new ErrorEvent('Network error', { message: 'message' }));
```

Component

Il est également important de tester nos composants visuels partagés (ne pas confondre les composants visuels avec containers qui sont eux moins intéressants à tester car ils ne servent qu'à intégrer les composants visuels et les services ensembles, il s'agit donc plus d'un travail de test e2e). Le test d'un composant est beaucoup plus fourni car il n'est pas uniquement composé de logique JavaScript mais également HTML :

- Récupérer des éléments du DOM :
  - fixture.debugElement.nativeElement.querySelector('span')
    - https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelector
  - fixture.debugElement.query(By.css('p'))
    - https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Selectors

  Vous pouvez alors, par exemple, appeler la méthode « click() » ou lire le contenu « .textContent ».
- Mettre à jour la vue d'après les changements appliqués dans le TS :
  - fixture.detectChanges()

```
describe('CalculatorComponent', () => {
  let component: CalculatorComponent;
  let fixture: ComponentFixture<CalculatorComponent>;
```

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ CalculatorComponent ]
  })
  .compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(CalculatorComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});

it('initialized with 0', () => {
  // Vérification que la valeur de base est vien 0
  expect(component.value).toBe(0);
  const result = fixture.debugElement.nativeElement.querySelector('span');
  // Vérification que la valeur affichée est vien 0
  expect(result.textContent).toBe('0');
});

it('can increment', () => {
  expect(component.value).toBe(0);
  // Récupération des différents éléments du DOM nécessaires au test
  const result = fixture.debugElement.nativeElement.querySelector('span');
  const button = fixture.debugElement.nativeElement.querySelector('#plus');
  // Click sur le bouton plus
  button.click();
  // Vérification que la valeur a bien été incrémentée
  expect(component.value).toBe(1);
  button.click();
  button.click();
  expect(component.value).toBe(3);
  // Mise à jour du DOM
  fixture.detectChanges();
  // Vérification que la valeur dans le DOM est correcte
  expect(result.textContent).toBe('3');
});

it('can decrement', () => {
  component.value = 5;
  expect(component.value).toBe(5);
  const result = fixture.debugElement.nativeElement.querySelector('span');
  const button = fixture.debugElement.nativeElement.querySelector('#minus');
  button.click();
  expect(component.value).toBe(4);
  button.click();
```

```
    button.click();
    expect(component.value).toBe(2);
    fixture.detectChanges();
    expect(result.textContent).toBe('2');
  });

});
```

Pour tester les services asynchrones qui ne rendent aucun résultat (comme la navigation), il est possible de vérifier leurs appels en utilisant un système d'espion :

```
  let component: CalculatorComponent;
  let router: Router;
  let fixture: ComponentFixture<CalculatorComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [ RouterModule.forRoot([]) ],
      declarations: [ CalculatorComponent ],
      providers: [ { provide: APP_BASE_HREF, useValue: '/' } ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(CalculatorComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
    // Récupérer l'instance du router
    router = getTestBed().get(Router);
  });

  it('redirect to other page', () => {
    const button = fixture.debugElement.nativeElement.querySelector('#navigate');
    // Ajouter un espion sur la méthode "navigateByUrl" du router
    const spy = spyOn(router, 'navigateByUrl');
    button.click();
    // Vérifiaction que la méthode a été appelée 1 fois
    expect(spy.calls.count()).toBe(1);
    // Récupération du premier paramètre fourni
    const url = spy.calls.first().args[0].toString();
    // Vérification que l'URL appelée est bien "/other"
    expect(url).toContain('/other');
  });
```

Pour simuler un @Input(), il faut simplement initialiser la valeur comme sur n'importe quelle classe avant d'appeler le premier « detectChanges() » :

```
  beforeEach(() => {
    fixture = TestBed.createComponent(CalculatorComponent);
    component = fixture.componentInstance;
    component.value = 10;
    fixture.detectChanges();
```

```
  });
```

Pipe

Pour tester les pipes, vous devez créer une instance de celui-ci et appeler ça méthode « transform » :

```
describe('Pipe: Date', () => {

  const pipe = new DatePipe();

  it('create an instance', () => {
    expect(pipe).toBeTruthy();
  });

  it('date to string', () => {
    const date = new Date();
    expect(pipe.transform(date)).toBe(date.toISOString());
  });
});
```

Test e2e

Il s'agit du contenu du dossier « e2e ».
Ces tests sont exécutables via la commande « ng e2e » (préférez « npm run e2e »).

Bonne nouvelle : Protractor ressemble fort aux tests unitaires, car il se base sur Karma. Il ne sera donc pas compliqué de se lancer.

Un test end-to-end se compose de deux fichiers : un « .po » (Page Object) qui contient les interactions avec l'application et un « .e2e-spec » qui contient les cas de test.

Page Object

Quand je parle d'interaction avec la page, je veux bien dire « toutes » les interactions. Exemples : les accès au contenu d'un span, le clic sur un bouton, écriture dans un champ, utilisation des touches spéciales comme les flèches…

```
export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('app-root h1')).getText();
  }

  clickOnNavigate() {
    return element(by.css('#navigate')).click();
  }

  selectNextKey() {
    browser.actions().sendKeys(Key.ARROW_RIGHT).perform();
  }
```

```
  writeElementLastname(lastname: string) {
    return element(by.id('lastname')).sendKeys(lastname);
  }
}
```

E2E

C'est dans cette étape que se trouve toute la logique des tests. Vous allez utiliser majoritairement les fonctions « toEqual », « toBe » ou encore « toBeTruthy » / « toBeFalsy ».

- **toBe** : test la référence ou la valeur des éléments primitifs (number, string, boolean)
- **toEqual** : test également la valeur des enfants et est donc utilisée principalement pour les objets et les tableaux
- **toBeTruthy** : vérifie si pas : null, undefined, 0 ou string vide
- **toBeFalsy** : le contraire de « toBeTruthy »

```
describe('workspace-project App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Welcome to testing!');
  });

  it('should navigate on click', () => {
    expect(page.getParagraphText()).toEqual('Welcome to testing!');
    page.clickOnNavigate();
    expect(page.getParagraphText()).toEqual('Page result');
  });

  it('should change element on right key click', () => {
    page.navigateTo();
    page.selectNextKey();
    expect(page.getParagraphText()).toEqual('selected 1');
    page.selectNextKey();
    expect(page.getParagraphText()).toEqual('selected 2');
  });

  it('should change paragraphe by name if name is entered', () => {
    page.navigateTo();
    const nameToWrite = 'Dupond';
    page.writeElementLastname(nameToWrite);
    expect(page.getParagraphText()).toBe(nameToWrite);
  });
});
```

Configuration

Vous trouverez le fichier de configuration « protractor.conf.js » également dans le dossier « e2e ».

Par défaut, la configuration est faite pour tourner sur la machine locale et tester uniquement sur Chrome une fois le projet buildé à l'aide de la commande « npm run e2e ».

Plusieurs navigateurs

Si vous voulez pouvoir tester sur différents navigateurs, vous devrez changer la ligne « capabilities » par « multipleCapabilities » :

```
capabilities: {
    'browserName': 'chrome'
  },
```

devient

```
multiCapabilities: [
    {
      'browserName': 'chrome',
    }
    …
  ],
```

Exécution à distance

Si vous voulez utiliser un hub selenium distant, il faut modifier « protractor.conf.js » mais également un peu la configuration du CLI.

Il faut tout d'abord modifier la ligne « baseUrl » pour mettre une adresse accessible depuis votre hub :

```
  baseUrl: 'https://monsite.azurewebsites.net',
```

Ensuite en spécifiant « directConnect » à « false », vous informez à protractor de ne pas lancer les navigateurs sur votre machine mais bien ceux distants :

```
  directConnect: false,
```

Enfin spécifiez l'adresse du hub selenium en ajoutant cette clé à la config :

```
 seleniumAddress: 'http://seleniumws2016.westeurope.cloudapp.azure.com:4444/wd/hub'
```

Pour finir allez dans le fichier « package.json » et modifiez le script « e2e » (pour ne pas lancer un serveur en local alors que nous testons une instance distante) par :

```
    "e2e": "ng e2e --serve=false"
```

Vous pouvez évidemment avoir un test protractor local et un distant en dupliquant la configuration « protractor.conf.js » et en utilisant cette nouvelle configuration de deux manières possibles :

- Soit en ajoutant **`--protractor-config`** dans un script du « package.json »
- Soit en copiant et adaptant la configuration actuelle « testing-e2e » dans le fichier « angular.json » et ajoutant **`--configuration`** dans un script du « package.json »